Load Packages

```
using TestImages, ImageShow, Colors, Statistics, PlutoTest, PlutoUI
```

`my_show`

```
my_show(arr::AbstractArray{<:Real}; set_one=false, set_zero=false)
```

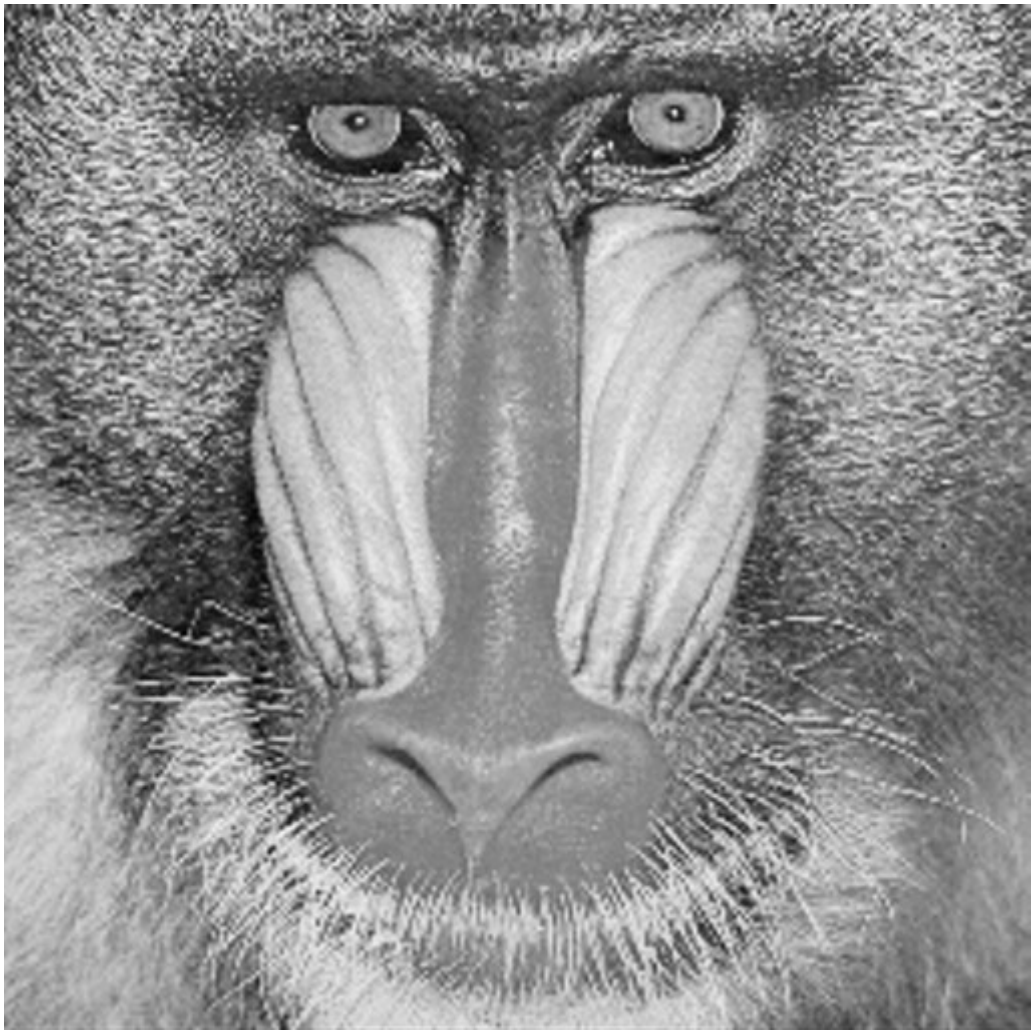
Displays a real valued array . Brightness encodes magnitude. Works within Jupyter and Pluto.

Keyword args

- `set_one=false` divides by the maximum to set maximum to 1
- `set_zero=false` subtracts the minimum to set minimum to 1

```
"""
    my_show(arr::AbstractArray{<:Real}; set_one=false, set_zero=false)
Displays a real valued array . Brightness encodes magnitude.
Works within Jupyter and Pluto.
## Keyword args
* `set_one=false` divides by the maximum to set maximum to 1
* `set_zero=false` subtracts the minimum to set minimum to 1
"""
function my_show(arr::AbstractArray{<:Real}; set_one=true, set_zero=false)
    arr = set_zero ? arr .- minimum(arr) : arr
    arr = set_one ? arr ./ maximum(arr) : arr
    Gray.(arr)
end
```

Homework 01



```
begin
    img = Float64.(testimage("mandril_gray"))
    my_show(img)
end
```

1. Add Noise To Images

This homework is dedicated to introduce into the basics of Julia. As examples, we want to apply and remove noise from images

Task 1.1 - Gaussian Noise

In the first step, we want to add Gaussian noise with a certain standard deviation σ. Try to get yourself used to the Julia documentation and check out whether such a function exists already and how you can apply it to your image/array.

Codewords: **Julia, random, random normal distribution**

add_gauss_noise

```
add_gauss_noise(img, σ=1)
```

This function adds normal distributed noise to `img`. `σ` is an optional argument

```
"""
    add_gauss_noise(img, σ=1)

This function adds normal distributed noise to `img`.
`σ` is an optional argument
"""
function add_gauss_noise(img, σ=one(eltype(img)))
    img = img .+ σ .* randn(size(img))
    return img
end
```

Now we want to use a for loop inside this function. Try to find out how to write for loops to iterate through an array.

add_gauss_noise_fl!

```
add_gauss_noise_fl!(img, σ=1)
```

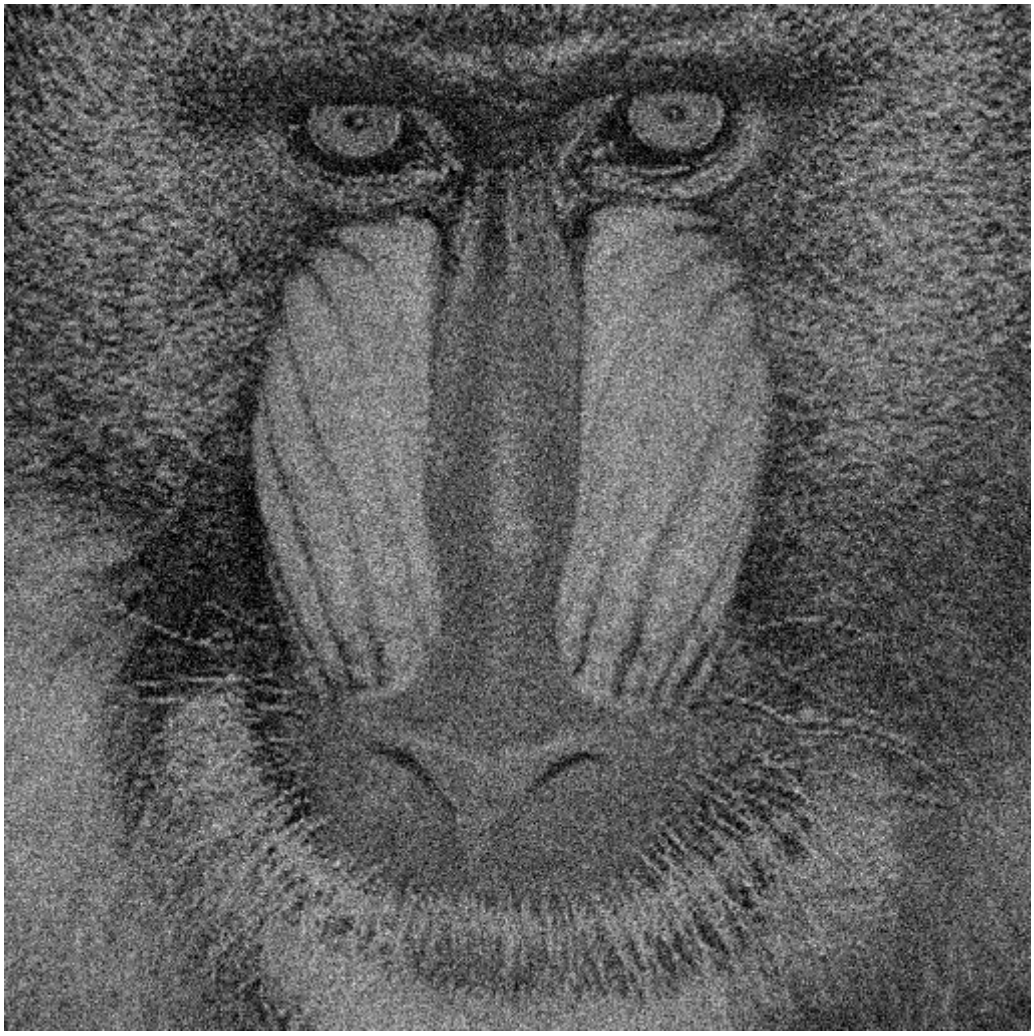
This function adds normal distributed noise to `img`. `σ` is an optional argument. This function is memory efficient by using for loops.

`!` means that the input (`img`) is modified. Therefore, don't return a new array but instead modify the existing one! The bang (!) is a convention in Julia that a function modifies the input.

```
"""
    add_gauss_noise_fl!(img, σ=1)

This function adds normal distributed noise to `img`.
`σ` is an optional argument.
This function is memory efficient by using for loops.


`!` means that the input (`img`) is modified.
Therefore, don't return a new array but instead modify the existing one!
The bang (!) is a convention in Julia that a function modifies the input.
"""
function add_gauss_noise_fl!(img, σ=1)
    img = img .+ σ .* randn(size(img))
end
```

Tests 1.1

Don't modify but rather take those tests as input whether you are correct. Green is excellent, red not :(



```
my_show(add_gauss_noise(img, 0.15), set_one=true)
```

Check wether mean and standard deviation are correct

```
≈(0.3, std(add_gauss_noise(ones((512, 512)), 0.3)), rtol = 0.1)
```

```
PlutoTest.@test ≈(0.3, std(add_gauss_noise(ones((512, 512)), 0.3)), rtol=0.1)
```

```
≈(0.3, std(add_gauss_noise_fl!(ones((512, 512)), 0.3)), rtol = 0.1)
```

```
PlutoTest.@test ≈(0.3, std(add_gauss_noise_fl!(ones((512, 512)), 0.3)), rtol=0.1)
```

```
≈(1, mean(add_gauss_noise(ones((512, 512)), 0.3)), rtol = 0.1)
```

```
PlutoTest.@test ≈(1, mean(add_gauss_noise(ones((512, 512)), 0.3)), rtol=0.1)
```

Task 1.2 - Poisson Noise

In microscopy and low light imaging situation, the dominant noise term is usually Poisson noise which we want to simulate here.

For adding Poisson noise we use: PoissonRandom.jl

Read the documentation of this package how to generate Poisson Random numbers

```
using PoissonRandom
```

87

```
pois_rand(100)
```

add_poisson_noise!

```
add_poisson_noise!(img, scale_to=nothing)
```

This function adds poisson distributed noise to `img`.

Before adding noise, it scales the maximum value to `scale_to` and divides by it afterwards. With that we can set the number of events (like a photon count).

Differently said: the `scale_to` applies Noise to an array equivalent to the noise level of an array with maximum peak `scale_to`.

If `isnothing(scale_to) == true`, we don't modify/scale the array.

`!` means that the input is modified.

```
"""
    add_poisson_noise!(img, scale_to=nothing)

This function adds poisson distributed noise to `img`.

Before adding noise, it scales the maximum value to `scale_to` and
divides by it afterwards.
With that we can set the number of events (like a photon count).

Differently said: the `scale_to` applies Noise to an array equivalent to the noise
level of an array with maximum peak `scale_to`.

If `isnothing(scale_to) == true`, we don't modify/scale the array.

`!` means that the input is modified.
"""
function add_poisson_noise!(img, scale_to=nothing)
    if isnothing(scale_to)
        for i in eachindex(img)
            img[i] = pois_rand(img[i])
        end
        return img
    else
        img .= img .* scale_to ./ maximum(img)
        for i in eachindex(img)
            img[i] = pois_rand(img[i])
        end
        img = img ./ scale_to
        return img
    end
end
```

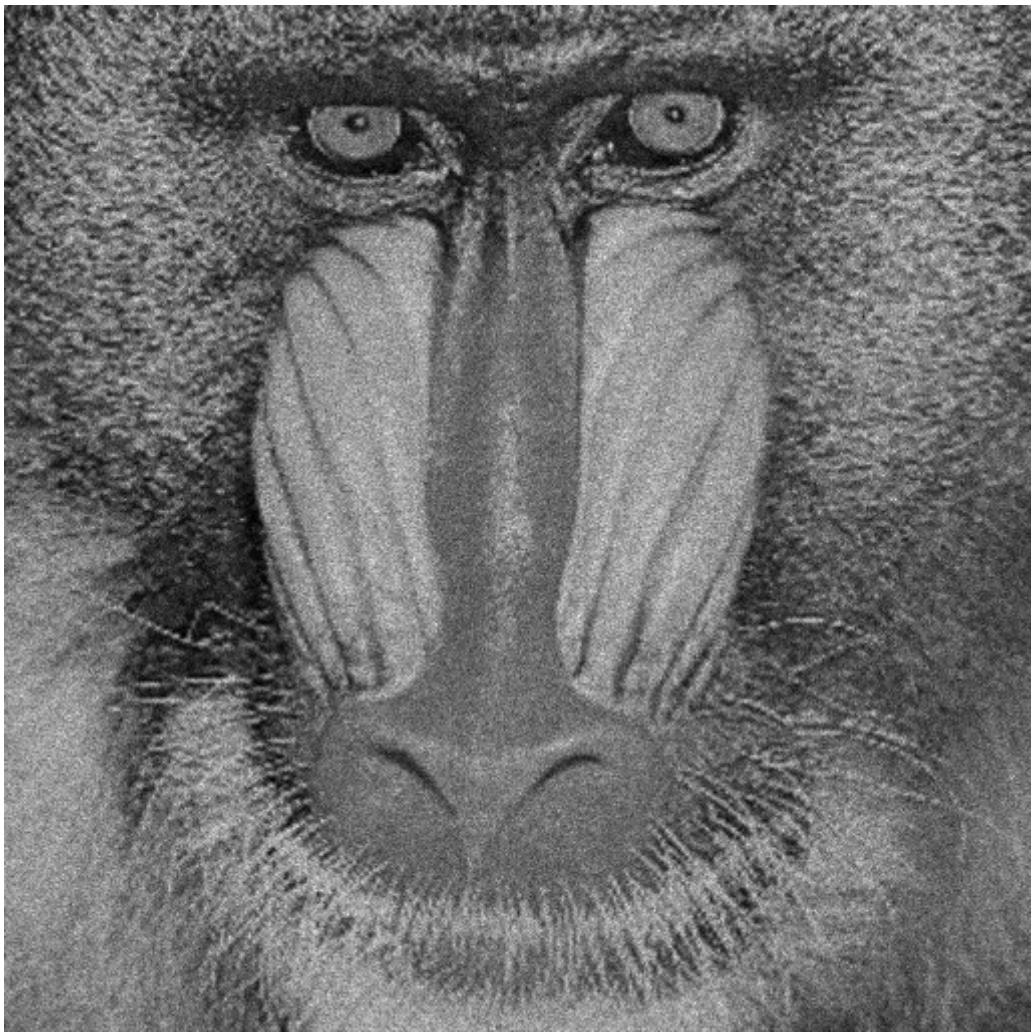
```
3×3 Matrix{Float64}:
 14.0  10.0  12.0
  8.0   8.0  10.0
  8.0  14.0  11.0
```

```
add_poisson_noise!(10 .* ones((3, 3)))
```

```
3×3 Matrix{Float64}:
 0.97  0.9   0.86
 1.15  0.94  1.1
 0.86  1.02  0.92
```

```
add_poisson_noise!(10 .* ones((3, 3)), 100)
```

Test 1.2 - Poisson Noise



```
my_show(add_poisson_noise!(100 .* img), set_one=true)
```

```
149.97414779663086
```

```
mean(add_poisson_noise!(150 .* ones(Float64, (512, 512))))
```

● ≈(150, mean(add_poisson_noise!(150 .* ones(Float64, (512, 512)))), rtol = 0.05)

```
PlutoTest.@test ≈(150, mean(add_poisson_noise!(150 .* ones(Float64, (512, 512)))), rtol=0.05)
```

● ≈(√150, std(add_poisson_noise!(150 * ones(Float64, (512, 512)))), rtol = 0.05)

```
PlutoTest.@test ≈(√(150), std(add_poisson_noise!(150 * ones(Float64, (512, 512)))), rtol=0.05)
```

Consider those tests as bonus

```
≈(150, 150 * mean(add_poisson_noise!(ones((512, 512)), 150)), rtol = 0.05)
```

```
PlutoTest.@test ≈(150, 150 * mean(add_poisson_noise!(ones((512, 512)), 150)),
rtol=0.05)
```

```
≈(√150, 150 * std(add_poisson_noise!(ones((512, 512)), 150)), rtol = 0.05)
```

```
PlutoTest.@test ≈(√(150), 150 * std(add_poisson_noise!(ones((512, 512)), 150)),
rtol=0.05)
```

1.3 Hot Pixels

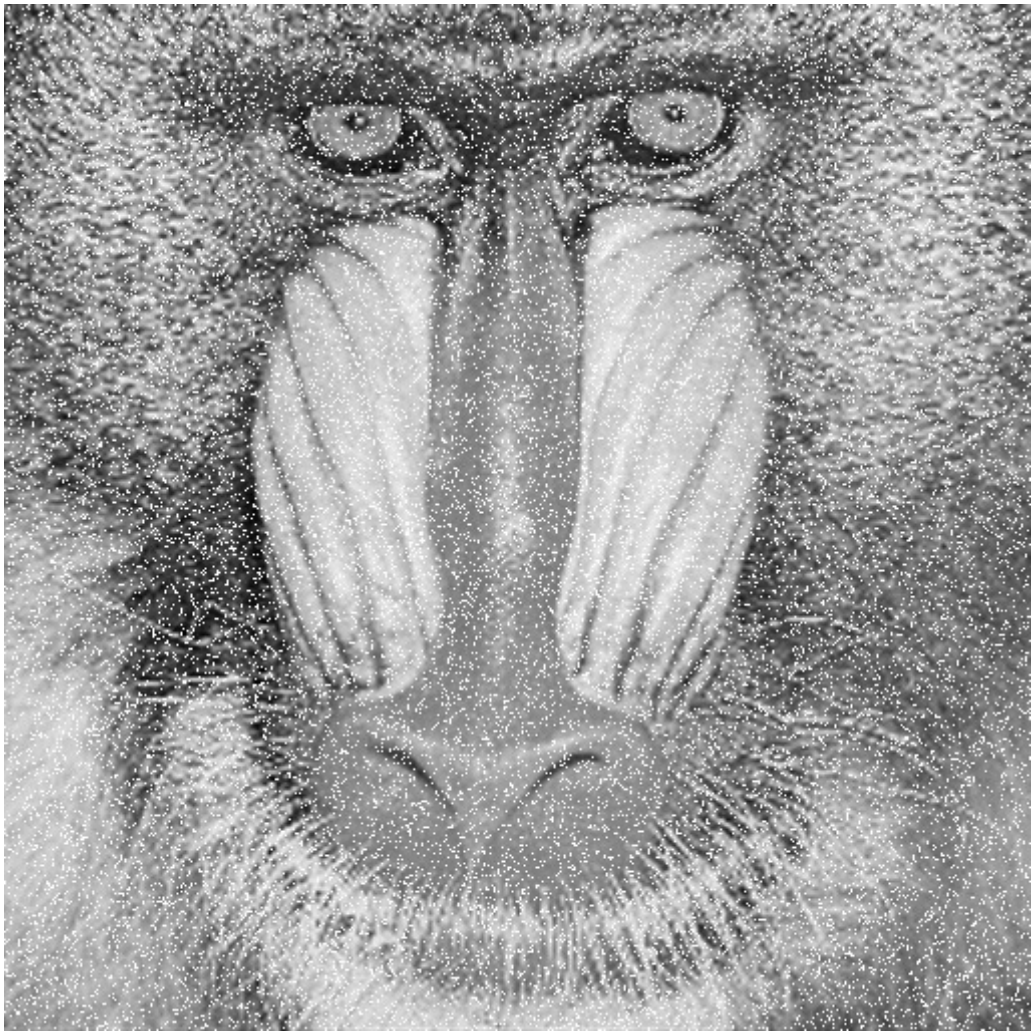
Another issue are hot pixels which result in a pixel with maximum value. This can be due to damaged pixels or some other noise (radioactivity, ...). Often this is called Salt (because of the maximum brightness) noise.

`add_hot_pixels!`

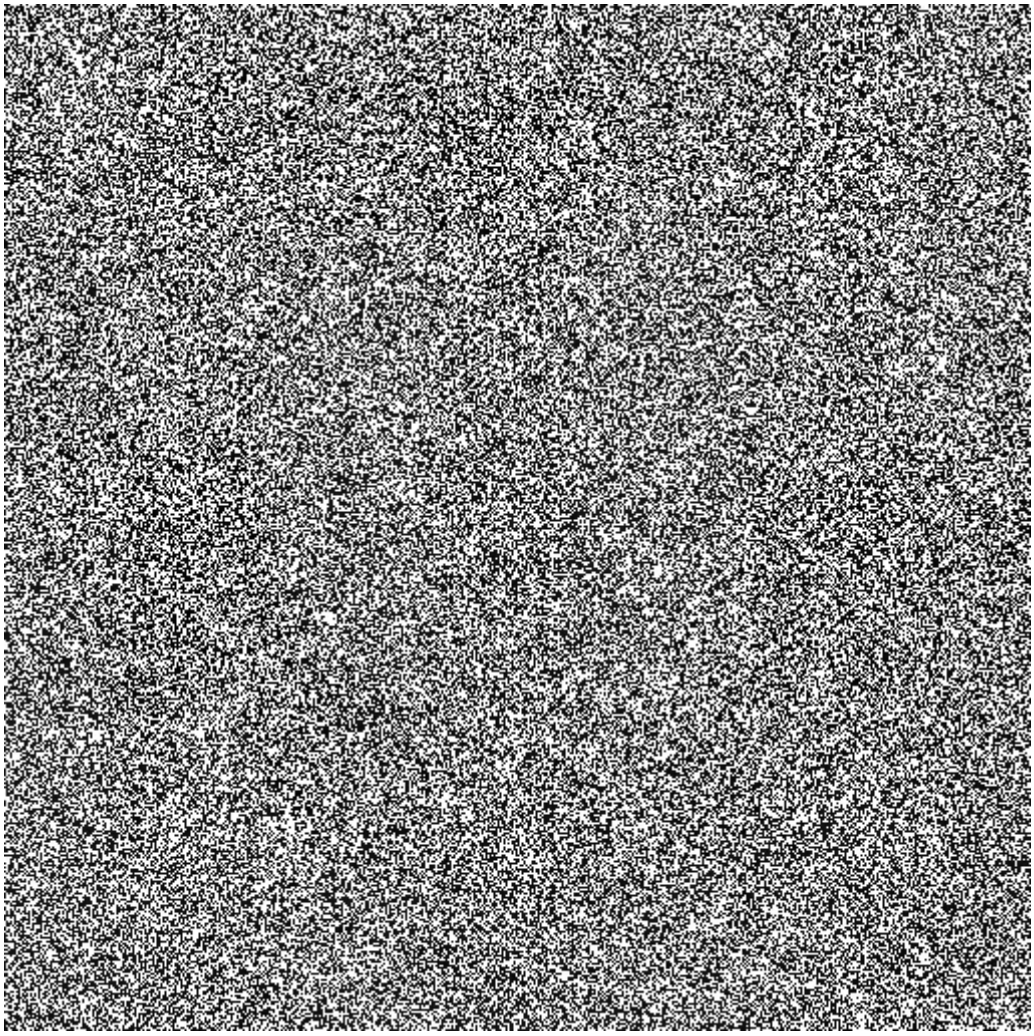
```
add_hot_pixels!(img, probability=0.1; max_value=one(eltype(img)))
```

Add randomly hot pixels. The probability for each pixel to be hot, should be specified by `probability`. `max_value` is a keyword argument which is the value the *hot* pixel will have.

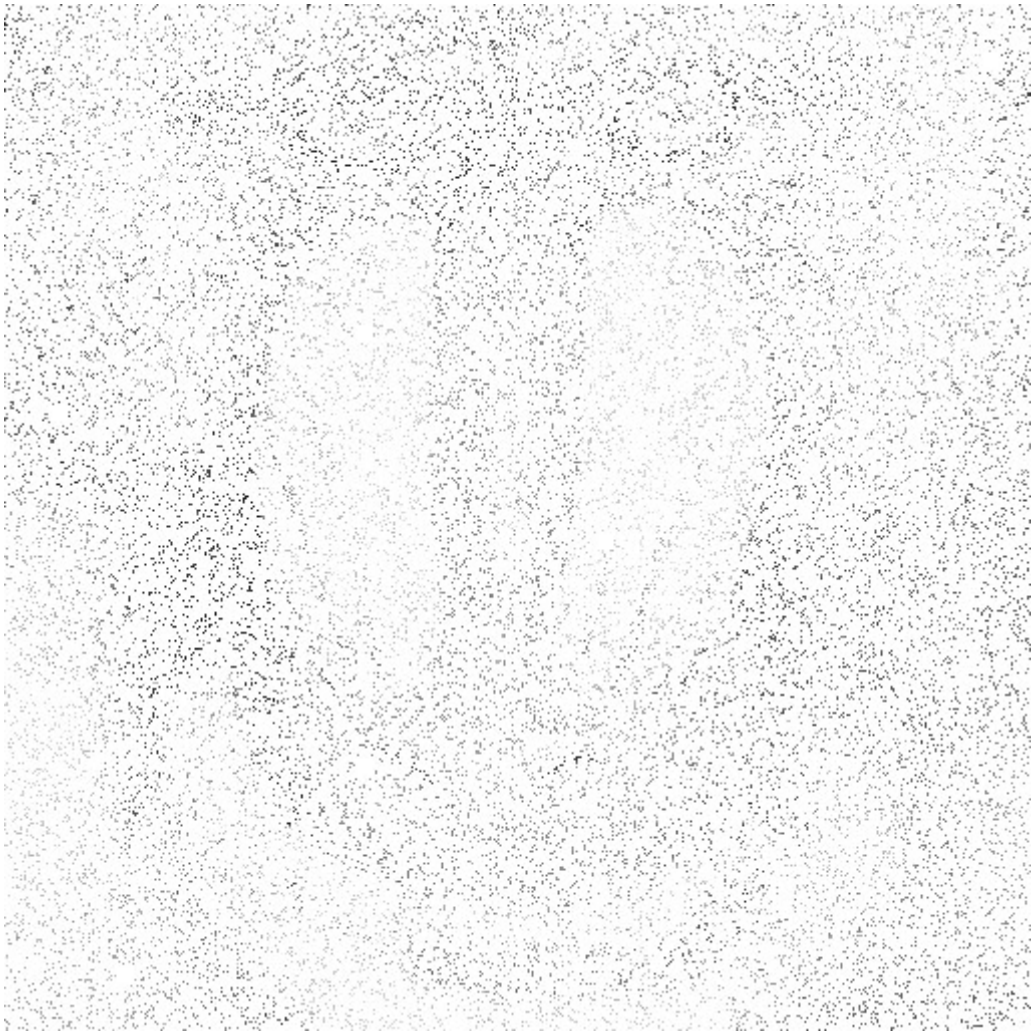
```
"""
    add_hot_pixels!(img, probability=0.1; max_value=one(eltype(img)))

Add randomly hot pixels. The probability for each pixel to be hot,
should be specified by `probability`.
`max_value` is a keyword argument which is the value the _hot_ pixel will have.
"""
function add_hot_pixels!(img, probability=0.1; max_value=1)
    img = img ./ maximum(img)
    for i in eachindex(img)
        if rand() < probability
            img[i] = max_value
        end
    end
    return img
end
```



```
my_show(add_hot_pixels!(copy(img)))
```



```
my_show(add_hot_pixels!(copy(img), 0.5; max_value=10))
```



```
my_show(add_hot_pixels!(copy(img), 0.9))
```

1.3 Test

```
≈(sum(map((x->x ≈ 2), add_hot_pixels!(copy(img), 0.5, max_value = 2))), length(img) .* 0.5, 1
```

```
PlutoTest.@test sum(map(x -> x ≈ 2, add_hot_pixels!(copy(img), 0.5, max_value=2))) ≈
length(img) .* 0.5 rtol=0.05
```

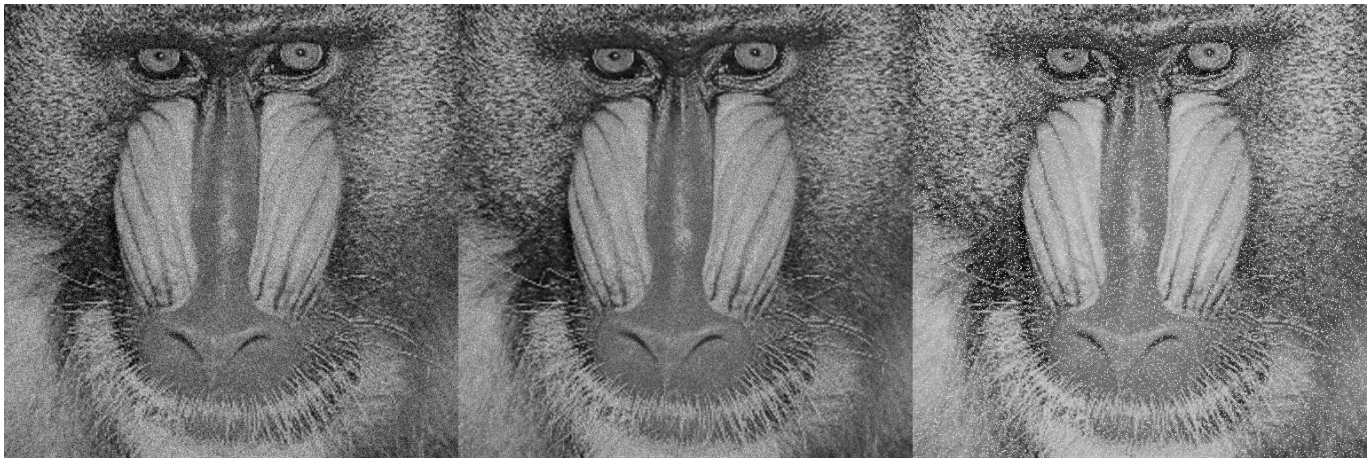
2. Remove Noise From Images

2.1 Remove Noise with Gaussian Blur Kernel

One option to remove noise is, is to blur the image with a Gaussian kernel. We can convolve a small gaussian (odd sized) Kernel over the array. For that we are using `ImageFiltering.imfilter` for the convolution.

To create a Gaussian shaped function, checkout `IndexFunArrays.normal`. You probably want to normalize the sum of it again

```
begin
    img_g = add_gauss_noise(img, 0.1)
    img_p = add_poisson_noise!(100 .* copy(img)) ./ 100
    img_h = add_hot_pixels!(copy(img))
end;
```



```
my_show([img_g img_p img_h])
```

```
using ImageFiltering, IndexFunArrays
```

```
imfilter (generic function with 11 methods)
```

```
# check out the help
imfilter
```

```
normal (generic function with 3 methods)
```

```
# check out the help
normal
```

```
gaussian_noise_remove (generic function with 1 method)
```

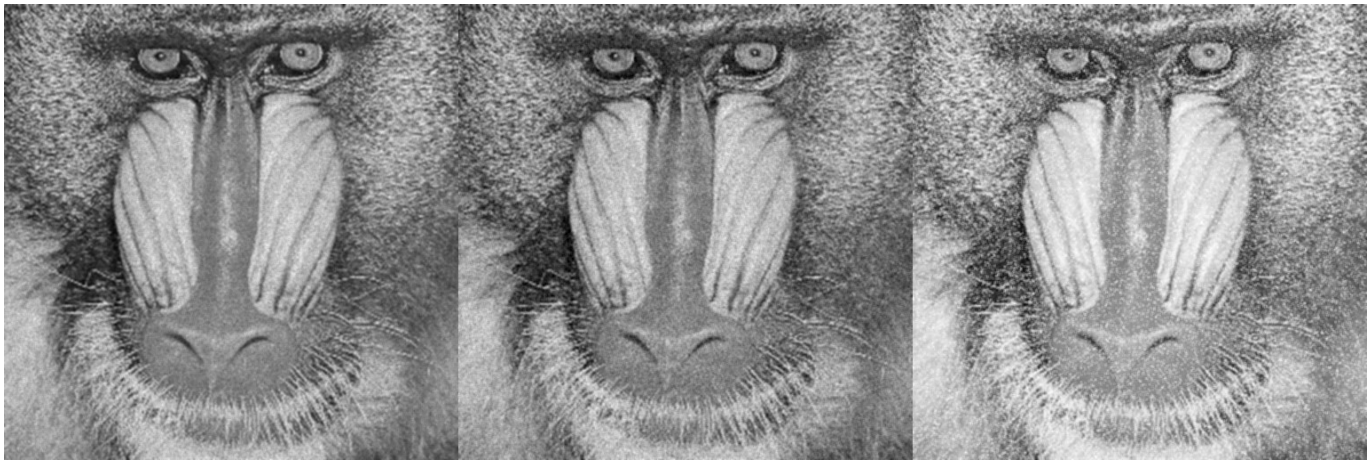
```
function gaussian_noise_remove(arr; kernel_size=(3,3), σ=1)
    gauss = IndexFunArrays.normal(kernel_size, sigma=σ)
    normalized_gauss = gauss ./ sum(gauss)
    filtered_arr = imfilter(arr, normalized_gauss)
    return filtered_arr
end

#This can pass the test when σ is larger than 0.11
```

σ = 2.91

```
md"σ = $(@bind σ Slider(0.01:0.1:10, show_value=true))"
```

kernel size = 3

```
begin
    img_p_gauss = gaussian_noise_remove(img_p, kernel_size=(ks,ks), σ=σ);
    img_g_gauss = gaussian_noise_remove(img_g, kernel_size=(ks,ks), σ=σ);
    img_h_gauss = gaussian_noise_remove(img_h, kernel_size=(ks,ks), σ=σ);
end;
```



```
my_show([img_p_gauss img_g_gauss img_h_gauss])
```

2.1 Test

```
arr_rand = add_gauss_noise(ones((500, 500)), 0.2);
```

std(gaussian_noise_remove(arr_rand, kernel_size = (8, 8), σ = 2)) < 0.05

```
PlutoTest.@test std(gaussian_noise_remove(arr_rand, kernel_size=(8,8), σ=2)) < 0.05
```

sum(abs2, img_g .- img) > sum(abs2, img_g_gauss .- img)

```
PlutoTest.@test sum(abs2, img_g .- img) > sum(abs2, img_g_gauss .- img)
```

2.2 Noise Removal with Median Filter

The median filter slides with a quadratic shaped box over the array and always takes the median value of this array.

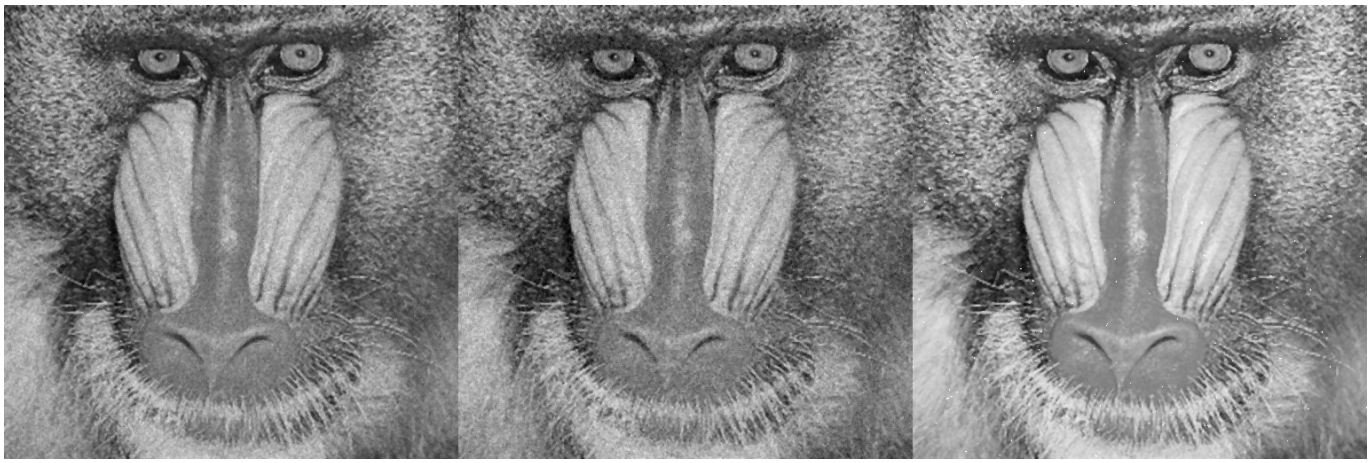
So conceptually, you can use two for loops over the first two dimensions of the array. Then extract a region around this point and calculate the median. Try to preserve the image output size. Assign the median to this pixel.

```
median_noise_remove! (generic function with 1 method)
```

```
#=
function median_noise_remove!(arr; kernel_size=(5,5))
    kx, ky = div(kernel_size[1], 2), div(kernel_size[2], 2)
    output = copy(arr)

    for i in 1:size(arr, 1)
        for j in 1: size(arr, 2)
            x_min = max(1, i - kx)
            x_max = min(size(arr, 1), i + kx)
            y_min = max(1, j -ky)
            y_max = min(size(arr,2), j + ky)

            window = arr[x_min:x_max, y_min:y_max]
            output[i,j] = median(window)
        end
    end
    arr .= output
    return arr
end
=#

function median_noise_remove!(arr; kernel_size=(5,5))
    removed_arr = mapwindow(median, arr, kernel_size)
    arr .= removed_arr
    return arr
end

# mapwindow is different from the loop above, and it can't pass the tests. But why it can pass the tests after the loop one runs?
```

kernel_size_2 = 3

```
begin
    img_p_median = median_noise_remove!(copy(img_p), kernel_size=(ks_2,ks_2));
    img_g_median = median_noise_remove!(copy(img_g), kernel_size=(ks_2,ks_2));
    img_h_median = median_noise_remove!(copy(img_h), kernel_size=(ks_2,ks_2));
end;
```



```
my_show([img_p_median img_g_median img_h_median])
```

2.2 Test

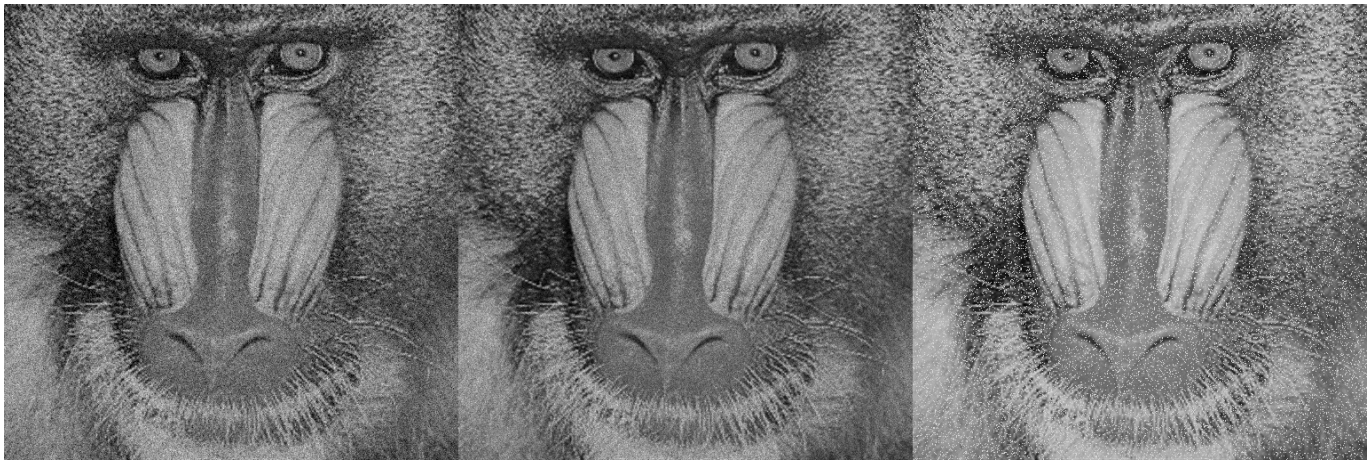
sum(abs2, img_p .- img) > sum(abs2, img_p_median .- img)

```
PlutoTest.@test sum(abs2, img_p .- img) > sum(abs2, img_p_median .- img)
```

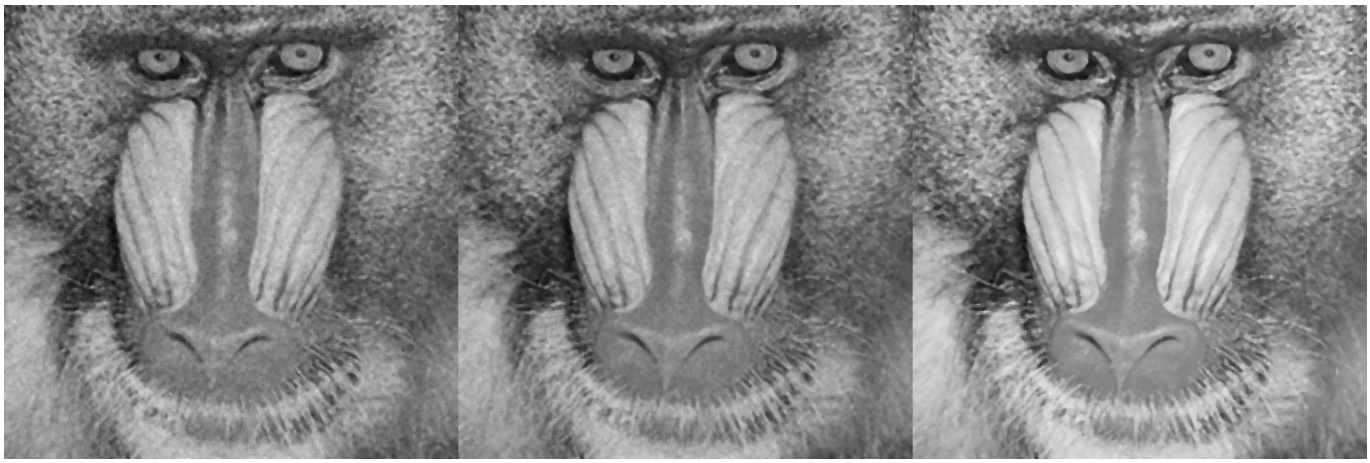
sum(abs2, img_g .- img) > sum(abs2, img_g_median .- img)

```
PlutoTest.@test sum(abs2, img_g .- img) > sum(abs2, img_g_median .- img)
```

3 Final Images



```
my_show([img_g img_p img_h])
```



```
my_show([median_noise_remove!(copy(img_g)) median_noise_remove!(copy(img_p))
median_noise_remove!(copy(img_h))])
```